# Are Automatically-Detected Code Anomalies Relevant to Architectural Modularity?
## An Exploratory Analysis of Evolving Systems

Isela Macia[1], Joshua Garcia[2], Daniel Popescu[2], Alessandro Garcia[1], Nenad Medvidovic[2], Arndt von Staa[1]

[1]Opus Group, LES, Informatics Department, PUC-Rio, RJ, Brazil
[2]University of Southern California, Los Angeles, CA, USA

{ibertran, afgarcia, arndt}@inf.puc-rio.br, {joshuaga, dpopescu, neno}@usc.edu

## ABSTRACT
As software systems are maintained, their architecture modularity often degrades through architectural erosion and drift. More directly, however, the modularity of software implementations degrades through the introduction of code anomalies, informally known as code smells. A number of strategies have been developed for supporting the automatic identification of implementation anomalies when only the source code is available. However, it is still unknown how reliable these strategies are when revealing code anomalies related to erosion and drift processes. In this paper, we present an exploratory analysis that investigates to what extent the automatically-detected code anomalies are related to problems that occur with an evolving system's architecture. We analyzed code anomaly occurrences in 38 versions of 5 applications using existing detection strategies. The outcome of our evaluation suggests that many of the code anomalies detected by the employed strategies were not related to architectural problems. Even worse, over 50% of the anomalies not observed by the employed techniques (false negatives) were found to be correlated with architectural problems.

***Categories and Subject Descriptors*** D.2.8 [Software Engineering]: Metrics; D.2.10 [Software Engineering]: Design; D.2.11 [Software Engineering]: Software Architectures.

***General Terms***: Measurement, Design.

***Keywords:*** Code anomalies; architectural degradation symptoms; architectural violations; architectural anomalies.

## 1. Introduction

Code anomalies, also referred in the literature as "code smells" [13], emerge in programs structured with any kind of modularization technique, including object-oriented programming [31] and aspect-oriented programming [19]. Code anomalies are often considered as key indicators of architectural degradation [13]. Hence, if these code anomalies are not systematically removed, the system's architectures may degrade due to erosion or drift [16]. Architectural erosion occurs when architectural violations are introduced, whereas drift is the realization of unintended design decisions also known as architectural anomalies [39].

The detection of architecturally-relevant code anomalies is particularly challenging when architectural designs are absent or obsolete, which is a common situation in evolving software projects. A complicating factor is that, due to time constraints, developers often need to concentrate on the most relevant anomalies. In other words, they should focus on code anomalies that are actually contributing to architecture erosion or drift. Let's consider a simple example of code anomaly, such as *God Class* [27]. Occurrences of *God Class* only cause harm to the architectural modularity when their realization of multiple concerns introduce undesirable dependencies between architecture elements (e.g., multiple architecture layers). Therefore, such *God Class* instances require closer, more immediate attention than other instances [37].

Recent research has developed complementary ways to improve automatic detection of code anomalies. They are usually based on exploiting information that is extracted from the source code [21, 26, 28, 32, 40, 46] and rely on the combination of static code metrics. These mechanisms, known as *detection strategies* [27], have been the subject of recent studies reported in the literature. Several studies have reported acceptable accuracy rates (60% or higher) for such strategies used in anomaly detection processes [27]. Studies have also evaluated the impact of the code anomalies detected by these strategies on maintenance effort [18, 37, 38]. However, it is still unknown whether the code anomalies detected by current strategies could be also used for indicating more severe architectural problems.

The objective of this paper is to assess the usefulness of automated code anomaly detection strategies for uncovering architecture modularity problems. To this end, we carried out an exploratory study to analyze the influences of code anomaly on architectural designs in 38 versions of 5 applications from heterogeneous domains. These


*AOSD'12*, March 25–30, 2012, Potsdam, Germany.

applications followed different architectural patterns and styles, such as Layers, Model-View-Controller and Aspectual Design [4]. The anomalies were detected in these systems using automated detection strategies [21, 27] which were the most effective to detect those anomalies in recent studies [18, 21, 23, 27, 44]. In addition, the original architects were consulted to reliably identify architectural problems in the studied systems. We explicitly selected systems for which architectural information was accessible so that we could correlate the architectural problems with the presence of code anomalies.

Our results confirmed that current automated strategies were not accurate to identify code anomalies that attempt against architectural modularity in the target systems. More specifically:

- More than 50% of the *automatically-detected code anomalies* were not correlated with architectural problems. This means that developers could spend most of the time reviewing code that might not represent threats to the system's architectural design.
- Even worse, more than 50% of the false negatives "generated" by automated strategies were found to be correlated with architectural modularity problems. This means that developers would lead to neglect code anomalies that are critical to architectural design.
- The inefficiency of detection strategies cannot be simply addressed by calibrating specific metric thresholds or determining different combinations of particular measures. It seems that their imperfection is largely due to their inability to exploit architectural concern's properties or architectural information in the source code.
- Certain recurring patterns of anomaly co-occurrences seem to be better indicators of architecture modularity problems than individual anomaly occurrences. These patterns usually cannot be directly specified and identified by existing detection strategies [21, 23, 32].

The rest of this paper is organized as follows. Section 2 presents the related work and its limitations in the context of this study. Section 3 introduces the different kinds code anomalies and of architectural modularity problems that are considered in our analysis. Section 4 describes our analysis procedures. Section 5 presents the obtained results, whereas Section 6 discusses their relevance and the main findings. Section 7 highlights the limitations of our study and, finally, Section 8 provides some concluding remarks.

## 2. Related Work

Our focus is on automatic detection of code anomalies as engineers usually do not have time and resources for carrying out a manual detection process. In this context, we divided the related work in two categories. First, we present current research aiming to support automatic detection of code anomalies. Second, we overview empirical studies that analyze the impact of automatically-detected code anomalies from different perspectives.

**Automatic Detection of Code Anomalies**. Emden and Moonen [9] describe *jCosmo*, an approach for detecting code anomalies based on the structural properties of code elements. Ratzinger et al. [41] detect code anomalies by examining change couplings. Strategies for detecting code anomalies [27] are the most common mechanism referenced and studied in the literature. The reason is that they generate a list of suspects; as a result, a wide range of static analysis tools, including visualization ones (e.g. [6, 49]), are based on such strategies. Marinescu [21, 27] introduced the concept of detection strategy, which consists of a logical expression composed by metrics to detect code anomalies. A concrete example is given in Section 3.1. Marinescu et al. [28] also presented *inCode*, a tool used to automate the application of certain detection strategies.

Other authors also proposed strategies and tools for anomaly detection in the literature. For instance, Munro [32] proposed some heuristics for detecting code anomalies. Alikacem and Sahraoui [1] proposed a language to detect code anomalies. This language allows the specification of rules using metrics and thresholds. Moha et al. [32] presented the *Decor*, a tool and a domain-specific language to automate the construction of anomaly detection strategies. However, none of these tools were used in the context of this work (Section 4.4) because they are neither available [1, 33] nor support the detection of all code anomalies that are considered in our analysis [28, 32].

**Studies of Code Anomalies**. The effectiveness of automatically-detected code anomalies using strategies have been recently studied under different perspectives. The authors have conducted empirical studies to investigate the negative effects of such automatically-detected anomalies. For instance, Mantyla and Lassensius investigate to what extent automatically-detected code anomalies can be used as a basis for subjective evaluation of code evolvability [25]. Olbrich et al. [37, 38] and Khomh et al. [18] investigate the evolution of automatically-detected code anomalies. The authors analyze whether the number of code anomalies increases over time, and the anomalies' influence on how often a code element changes. However, they did not study to what extent these phenomena are related to architectural modularity problems (Section 3). Certain classes (classified as "anomalous") might coincidently change more often because the associated requirements are naturally more volatile than others. In other words, the rate of individual class changes might not necessarily be an indicator of architecture modularity problems.

Other works investigate the impact of automatically-detected code anomalies on software defects (i.e. the need for corrective maintenance). For instance, D'Ambros et al. [6] found that, while some code anomalies are more frequent, none of them can be considered more harmful with respect to software defects. In the context of aspect-oriented systems, Macia et al. [23] analyzed the influence of code anomalies on corrective changes. They also analyzed their influence on perfective changes (i.e. refactoring effort).

However, none of these analyses investigate to what extent detection strategies accurately localize code anomalies that are related to architectural modularity problems. That is, they do not assess the impact of automatically-detected code anomalies on architectural designs.

## 3. Code Anomalies and Architectural Problems

This section introduces relevant concepts related to code anomalies (Section 3.1) and architectural modularity problems (Section 3.2). It also illustrates how they could be interrelated (Section 3.3) using a running example.

### 3.1 Code Anomalies and Detection Strategies

The modularity of system implementations degrades through the introduction of code anomalies. They affect different code units, such as classes and methods. For example, *God Class* is a code anomaly in which a class (i) realizes various concerns – i.e. it performs too much work on its own, delegating minor responsibilities to a set of simple classes, and (ii) uses data from many other classes, increasing its coupling. For instance, *MediaController* in Figure 1 was classified as a *God Class* instance.

Detection strategies interpret a set of code metrics that are extracted from a specific code element (e.g., class or method) by using a set of threshold filter rules [27]. Then, the results of these filters are combined and used to identify code anomalies. Below, we present the detection strategy used to identify *God Classes* in our study. This detection strategy and its thresholds were defined in [21] and have been used in other studies [37, 38]. The calibration of these thresholds (see Section 4.4) is required in some cases.

$$GodClass(C) = (WMC(C) \geq 47) \wedge (TCC(C) < 0.3) \wedge (ATFD(C) > 5)$$

where:

- C is the class being inspected;
- Weighted Method Count (WMC(C)) is the sum of the cyclomatic complexity of all methods in C [21];
- Tight Class Cohesion (TCC(C)) is the relative number of directly connected methods, i.e. methods that access the same instance variables, in C [30];
- Access To Foreign Data (ATFD(C)) is the number of attributes in foreign classes accessed by class C [21].

Table 1 summarizes the set of code anomalies that we have analyzed in this study. These anomalies were selected because they represent all the code anomalies identified by developers in the systems we analyzed (Section 4.3). In addition, their detection strategies have been widely studied (Section 2), with detection accuracy rates higher than 60%. The details governing the anomalies' definitions as well as their detection strategies can be found in [21, 23, 44]; we elide them here for brevity and space constraints.

### 3.2 Degradation: Architecture Modularity Problems

The phenomenon of architectural degradation was introduced by Hochstein and Lindvall [16] as aiming at

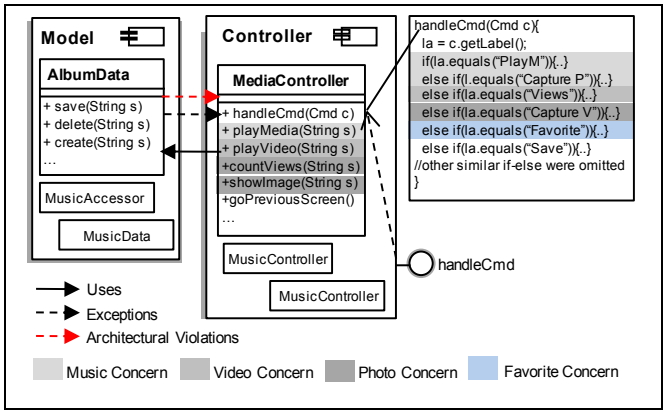


Figure 1. Relationship between code anomalies and architectural modularity problems

referring to the continuous decline of the architectural modularity. Architectural degradation encompasses architectural modularity problems caused by the processes of erosion and drift [39].

**Architectural Erosion** is the process of introducing decisions into a system architectural design that violates the system's intended architecture [39] and, therefore, attempt against the architectural modularity. As a result, architecture violations can only be observed if an explicit specification of the prescribed (i.e. intended) architectural design decisions is available.

Figure 1 depicts an example of architectural violation in the MobileMedia system. The problem is associated with the realization of the exception handling policy in this system. Most exceptions are propagated through component interfaces across the system layers, thereby going against the architects' original intent in some cases. For instance, the MediaController component invokes different services from the *AlbumData* component, which partially realizes the *Model* layer. *MediaController* ends up handling exceptions (e.g. *PersistentException*) signalized by *AlbumData*, including those that should be treated internally to the other component realizing the *Model* layer. This means that additional code couplings, between elements realizing the *Model* and *Controller* layers, are the sources of architecture violations.

**Architectural Drift** is the introduction of design decisions into a system's architecture that were not included in the intended architecture, albeit they do not violate any of the

Table 1. Code anomalies analyzed in our study

| Aspect-Oriented Code Anomalies | Object-Oriented Code Anomalies |
|---|---|
| Composition Bloat [23] | Data Class [13, 21] |
| Duplicate Pointcut [44] | Divergent Change [13, 21] |
| Forced Join Point [23] | Feature Envy [13, 21] |
| God Aspect [23] | God Class [21, 29] |
| God Pointcut [23] | Large Class [13, 21] |
| Redundant Pointcut [23] | Long Method [13, 21] |
| | Long Parameter List [13, 21] |
| | Misplaced Class [13, 21] |
| | Shotgun Surgery [13, 21] |
| | Small Class [13, 21] |

prescribed design decisions [39]. Some architectural drift symptoms are caused by applying a design decision that neglects or impairs one or more modularity principles. Each of these symptoms of architectural drift is often referred as an *architectural anomaly* [14, 22]. These anomalies comprise decisions that may negatively impact architectural modularity principles, such as narrow component interfaces and components realizing a single concern [14, 29].

In order to select a set of drift symptoms to be analyzed in our study, we considered catalogs of architectural anomalies explicitly documented in the literature [14, 29]. Our final subset of analyzed anomalies encompassed those that were identified by architects in the target systems of our study (Section 4.3). The types of architectural anomalies analyzed in our study are summarized in Table 2. Note that each architectural anomaly hinders different modularity principles. For instance, while *Component Concern Overload* anomaly does not adhere to the single responsibility principle, the *Ambiguous Interface* violates the simple interface principle [29].

Table 2. Architectural anomalies detected in our study

| Architectural Anomalies | Definition |
|---|---|
| Ambiguous Interface | Interfaces that offer only a general entry-point into a component that handles more requests than it should actually process |
| Extraneous Connector | Connectors of different types are used to link a pair of components |
| Connector Envy | Components which realize functionality that should be assigned to a connector |
| Scattered Parasitic Functionality | Multiple components are responsible for realizing the same high-level concern and orthogonal ones |
| Component Concern Overload | Components responsible for realizing two or more unrelated architectural concerns |

As an illustration, Figure 1 depicts an *Ambiguous Interface* anomaly in the context of the MobileMedia system used in our study [12]. The interface *handleCmd* of the component *Controller* is implemented by the class *MediaController*. However, the interface offers only one method, which receives all service requests and, therefore, it handles more types of commands than it should actually process; i.e. it receives the parameter *Cmd* with the generic type *Command*. This situation hinders the architecture modularity as over-generalized interfaces allow additional dependencies between components [14, 29]. Note that even though this situation is not necessarily a violation of prescribed design decisions (i.e. symptoms of architecture erosion), over-generalized interfaces might favor tight component coupling as the system evolves. Additional examples and a discussion of each architectural anomaly can be found in [14, 29] and is out of the scope of our work.

### 3.3 Code Anomalies as Indicators of Architecture Degradation Symptoms

Previous research [21, 32] departs from the following assumption: detection strategies (Section 3.1) accurately localize code anomalies related to architectural modularity problems (Section 3.2). Our study (Section 4) is aimed to analyze to what extent this assumption holds. A code anomaly *C* is related to an architecture modularity problem *A* when: (i) the code elements (e.g., methods or classes) affected by *C* are in charge of implementing the architectural elements (e.g., components, interfaces, connectors), and (ii) these architectural elements are affected by *A*. In this work we considered only those relationships for which the cause-effect was confirmed by architects (Section 4.4).

Figure 1 depicts an example of this cause-effect relationship. The *MediaController.handleCmd* method was considered as the source of two code anomalies and one architectural anomaly. First, this method was classified a *Long Method* (Table 1) as it contains many lines of code, presents high cyclomatic complexity, and realizes several architectural concerns. This method was also classified as *Divergent Change* because it is using information from several classes to deal with different services. In addition, its implementation is responsible for dealing with different exceptions propagated by the *Model* component; however, these exceptions are not related with the method's goal. Finally, this method is in charge of implementing the interface *handleCmd* of *Controller* classified as *Ambiguous Interface*.

## 4. Study Definition and Design

The relationship between code anomalies and architectural modularity problems has often been recognized by the literature [13, 16]. However, as software projects evolve, the source code is usually the only artifact available and the architectural design is not explicitly documented. Hence, the detection of code anomalies related to architectural problems is only viable if the available automated strategies (Section 3.1) are accurate. This study aims at analyzing whether detection strategies are accurate in localizing architecturally-relevant anomalies in the source code.

### 4.1 Hypotheses

In order to evaluate the accuracy of detection strategies for localizing architectural modularity problems (Section 3.2), we have divided the analysis into two perspectives by observing both architectural violations and architectural anomalies. These perspectives lead us to two null hypotheses H1 and H2 as defined below.

**$H1_0$**: *The accuracy of detection strategies to identify code anomalies related to architectural violations is high*.

**$H2_0$**: *The accuracy of detection strategies to identify code anomalies related to architectural anomalies is high.*

Strategies are considered accurate in the literature when their precision and recall rates are 60% or higher for detecting code anomalies [18, 27, 37, 38]. This threshold has been derived from empirical studies involving systems implemented using different programming languages [21].

As it is assumed that code anomalies are intrinsically related to architectural modularity problems [13, 16], we have used the same threshold in this study for assessing the strategies' accuracy to localize architecturally-relevant code anomalies. That is, we consider that their accuracy was 'high' if 60% of the code anomalies related to architectural modularity problems (Section 3.3) are automatically-detected.

### 4.2 Variable Selection

In order to test our hypotheses, we have defined the following independent and dependent variables.

**Independent Variable**. There are as many independent variables as there are kinds of automatically-detected code anomalies (Table 1). Each variable $C_{i,k,j}$ indicates the number of times that an entity $i$ suffers from a code anomaly $k$ in a version $v_j$. All code anomaly occurrences used in testing these hypotheses were confirmed by developers (Section 4.5).

**Dependent Variable**. Similar to the independent variables, there are many dependent variables as there are kinds of code anomalies. The dependent variables $V_{i,k,j}$ and $A_{i,k,j}$ for H1 and H2 indicate whether the entity $i$ affected by the code anomaly $k$ is introducing any violation or architectural anomaly in a version $v_j$, respectively. All instances of architectural degradation symptoms used in testing these hypotheses were confirmed by the original architects (Section 4.4).

### 4.3 Target Systems

In this study we decided to focus on investigating short-term architectural modularity problems because they can provide early symptoms of architectural degeneration. For this kind of study, it is important to select systems implemented with object-oriented programming and aspect-oriented programming. The goal is to make a broader analysis and identify whether there could be any influence of the modular programming technique on the results. However, the comparison between the strategies' accuracy rates for these two programming techniques is beyond the scope of our study. It is also important to select systems developed using different practices related to architectural rule enforcement in the source code as well as counting on the availability of their original architects and developers. Their availability is important to help us to validate the identified architectural modularity problems (Section 3.3). A complete list of criteria for supporting the system selection process is provided in [5].

Based on the aforementioned criteria, we chose 38 releases of 5 medium-sized applications. Table 3 summarizes the general characteristics of each target system. Two of these applications are Web-based information systems, which allow citizens to register complaints about health issues in public institutions. HealthWatcher (HW) [15] is based on the layers architecture style. AspectualWatcher (AW) also follows this style, but relies on aspect-oriented design [4] for modularizing concerns that crosscut the layers in the HW system. Note that in this table the token "/" is used to separate the data of the object-oriented (Java) version and its aspect-oriented (AspectJ) counterpart.

We have also selected two software product lines and a middleware. The third and fourth systems are product lines for deriving applications that manipulate media on mobile devices [12]. MobileMedia (MM), relies on the model-view-controller architectural pattern, while AspectualMedia (AM) was structured based on aspectual architecture design for modularizing features that crosscut the MM architecture. The fifth system is a lightweight middleware platform, called MIDAS, for distributed, event-based sensor applications [24]. The two selected versions are the before and after versions of a major architectural restructuring with the widest impact in this system history. A high number of changes of architectural elements took place in this transition and are realized by the latest version.

Table 3. Systems used in our study

| | HW/AW | MIDAS | MM/AM |
|---|---|---|---|
| Application Type | Web-based system | Middleware | Software Product Line |
| Code Availability | Java/Aspect | C++ | Java/Aspect |
| # of Versions | 10/10 | 2 | 8/8 |
| # of Selected Versions | 10/10 | 2 | 8/8 |
| Avg. # of CE | 85/113 | 22 | 60/94 |
| Avg. # of AE | 34/41 | 14 | 48/61 |
| Avg. KLOC | 6 | 7 | 8 |

HW=HealthWatcher; AH=AspectualWatcher; MM=MobileMedia; AM=AspectualMedia; CE=Code Elements (classes and aspects); AE=Architectural Elements (components and connectors)

### 4.4 Procedures for Data Collection

In order to perform the data collection process we count on the help of two groups of architects: (i) those that defined the original intended architecture, and (ii) independent reviewers of the software architecture; and on a group of original developers. These three groups were involved in the main phases of our study, which are described next.

**Recovering the Actual Architecture**. This phase was based on a semi-automatic process. We have used Sonar [43] and Understand [47] to support the recovery of the actual architecture from the source code. These tools support architecture and code analyses in order to help developers to analyze and measure the modularity of the system's architecture and implementation. To make possible the architectural analysis, architects and original developers mapped code elements to architectural elements. These mappings allowed us to trace the influence of a code anomaly on the introduction of modularity problems in a system's architecture. These mappings also allowed us to identify how modularization of architectural concerns in the code were related to architecture modularity problems. An example of this mapping is showed in Figure 1 where the *MediaController.handleCmd* method is implementing *Music*, *Video*, *Photo* and *Favorite* concerns.

**Identifying Architectural Degradation Symptoms**. In order to identify symptoms of architectural erosion we used Software Reflexion Model [35]. As this technique demands the intended architecture was provided by architects. The comparison of the actual, extracted architecture (EA), and the intended architecture (IA) was supported by the two groups of architects. They were responsible for measuring the architecture conformance in terms of *convergence* (a component or relationship that is in both EA and IE), *divergence* (a component or relationship that is in EA but not in IA), and *absence* (a component or relationship that is in IA but not EA). For instance, all absence classifications were considered as violations. Although divergence classifications are natural suspects of possible violations, they can be related to unintended architectural decisions. Therefore, architects needed to validate their actual impact on architecture designs.

Furthermore, architectural anomalies were detected by architects based mainly on: (i) a visual inspection of the EA, and (ii) a careful analysis of the code elements mapped to architectural elements, due to the lack of tools. We also asked the architects to indicate other anomalies observed in the architecture design beyond those presented in Table 2. This helped us to better judge whether and which code anomalies are good indicators of architectural modularity problems.

As result of this stage, architects provided reports describing the architectural problems observed in each system's version. These reports described, for instance, the problem's type (e.g. violation, architectural anomaly), its location in the design, the architectural elements related to it and, in some cases, an explanation of the problem's cause.

**Automatic Detection of Code Anomalies**. Code anomalies were automatically identified using detection strategies. We selected metrics and thresholds that have shown high accuracy to identify code anomalies in previous studies [21, 23]. Sometimes, the thresholds suffered some minor adjustments in order to maximize the accuracy. For instance, certain thresholds were calibrated according to the specific programming styles and system characteristics [20]. When multiples detection strategies for a code anomaly were available in the literature, we analyzed which metrics and thresholds would be the most appropriate to reach the highest accuracy rates. The goal was to get the best possible results with the detection strategies at hand. If needed, the changes in the original detection strategies [21, 23, 44] were discussed with the systems' original developers. A complete list of the detection strategies used and their corresponding thresholds are available in a supplementary web site [5].

Furthermore, the metrics used in the detection strategies were mostly collected with existing tools such as: MuLATo [34], Together [45] and Understand [47]. These tools were chosen as they are complementary: MuLATo is a static analyzer for AspectJ programs whereas Together and Understand analyze Java programs. They have been used in previous studies reported in the literature [15, 21, 23] and, more importantly, they collect a large number of metrics that were required for the detection strategies employed.

### 4.5 Analysis Method

We also asked the developers to identify all the code anomalies that influenced on the architectural design. The lists of code anomalies provided by developers included fine-grained and accurate details about the code anomaly facilitating our analysis. For instance, the lists describe the code anomaly's type, the code elements affected by it, and its correlation with the architectural problems previously identified by architects. Afterwards, a stage was dedicated to investigate the accuracy of the detection strategies [21, 23] when detecting the code anomalies previously identified by developers. Therefore, this investigation was based on both lists: (i) automatically-detected code anomalies using existing detection techniques and, (ii) code anomalies detected by developers through the code review stage. In particular, the lists provided by developers were useful to assess the impact of non-automatically-detected code anomalies on architectural decompositions.

In order to reject $H1_0$ and $H2_0$, we calculated the precision and recall of detection strategies using the following formulas:

$$precision = \frac{TP}{TP + FP} \qquad recall = \frac{TP}{TP + FN}$$

where, *True Positive* (TP) and *False Positive* (FP) encompass all automatically-detected code anomalies that respectively were or not confirmed as relevant by architects and developers. As we described previously developers performed a code review in order to detect code anomalies related to architectural problems that were not automatically identified by the detection strategies; i.e. *False Negative* (FN). Based on these criteria, a detection strategy achieves 100% of precision and 100% of recall if it only pinpoints the same set of architecturally-relevant code anomalies confirmed by developers.

## 5. Study Results

Before discussing the strategies' accuracy to identify architecturally-relevant code anomalies (Section 5.2), this section presents how often the code anomalies were actually related to architecture problems in the target systems. Tables 4 and 5 summarize the overall impact of code anomalies on architectural designs. The list of code anomalies (used to compute the table values) represents all the occurrences of anomalies (Section 3.2), whether automatically detected by the strategies or not. The tables present, for each of the target systems, the relationship between code anomalies and architectural violations (Table 4) or particular types of architectural anomalies (Table 5). The columns are headed with the acronym of each system. The rows x and S in both tables represent the mean and the standard deviation, respectively. Violations in Table 4 were related to divergent relationships (Section 4.4) as the system's implementation

started based on its intended architecture. Data for MIDAS are not presented in Table 4 as no violation occurred in this system. This observation was expected as the development process in the MIDAS project strictly enforced architecture conformance [24] .

Table 4. Code anomalies related to architectural violations

| | | AW | HW | AM | MM |
|---|---|---|---|---|---|
| ***Violations*** | **X** | 134.9 | 207.2 | 43.5 | 46.7 |
| | **S** | 2.81 | 9.46 | 4.57 | 4.56 |
| ***Non-related*** | **X** | 24.7 | 51.8 | 10.87 | 7.8 |
| | **S** | 1.82 | 5.12 | 2.29 | 4.67 |
| **Total** | | **160** | **259** | **54** | **55** |

Table 5. Code anomalies related to architectural anomalies

| | | AW | HW | AM | MM | MIDAS |
|---|---|---|---|---|---|---|
| ***Ambiguous Interface*** | ***X*** | 8.6 | 6.4 | 9.25 | 12.16 | 2.5 |
| | ***S*** | 2.06 | 1.03 | 2.48 | 3.22 | 0 |
| ***Connector Envy*** | ***X*** | 5.6 | 5.6 | 7.25 | 8.63 | 2 |
| | ***S*** | 2.75 | 2.75 | 2.17 | 2.95 | 0 |
| ***Component Concern Overload*** | ***X*** | - | 3 | 1.41 | 2.73 | 1 |
| | ***S*** | - | 0 | 2.39 | 1.86 | 0 |
| ***Extraneous Connector*** | ***X*** | 3.8 | - | 1.38 | - | 14 |
| | ***S*** | 1.82 | - | 0.84 | - | 3.18 |
| ***Scattered Parasitic Functionality*** | ***X*** | 4.7 | 2.2 | 2.13 | 3.75 | - |
| | ***S*** | 1.85 | 1.01 | 1.18 | 1.3 | - |
| ***Non-related*** | ***X*** | 3.1 | 2.7 | 1.6 | 2.62 | 6 |
| | ***S*** | 1.44 | 1.91 | 1.52 | 1.78 | 4.5 |
| **Total** | | **26** | **20** | **22** | **30** | **29** |

A first analysis of Tables 4 and 5 revealed that the architectural modularity problems were significantly related to code anomalies. The correlation was usually higher than 80% for both violations and architectural anomalies. This conclusion can be drawn by comparing the total number of architectural modularity problems (row "Total") and the mean of those problems unrelated to code anomalies (i.e. row "Non-related") in each table. Interestingly, around 15% of the architectural modularity problems were related to code anomalies that emerged in the first system's versions. On the other hand, less than 20% of the architectural problems were not related to code anomalies. From the opposite perspective, we observed that just about 10% of the architectural anomalies were not related to code anomalies.

The aforementioned results were particularly relevant as the high correlation coefficient was observed even in systems developed with modularity principles in mind. The developers tried to maximize such principles in both architecture design and implementation phases. These results confirm that code anomalies may be indicators of architectural modularity problems in the source code. It reinforces the motivation of using detection strategies as indicators of architectural modularity problems in the source code. On the other hand, the success of this approach largely depends on the accuracy of existing strategies to detect architecturally-relevant code anomalies.

### 5.1 Diverse Degradation Symptoms in the Systems

The individual analysis of the systems revealed that the HealthWatcher (HW) system presented the largest number of architectural violations of the five systems. The number of violations increased over time in this system, leading to the highest architecture erosion rate. According to its architects, the main reason for introducing violations was the incremental addition of classes in the *GUIElements* layer that illegally access information in the *DataManagement* layer.

On the other hand, the MobileMedia (MM) system presented the largest number of architectural modularity anomalies of the five systems. The majority of these architectural drift symptoms were related to code anomalies that emerged along the system evolution. In particular, they were mostly caused by the non-modular realization of new concerns progressively included in the latest system versions. They were often instances of the following architectural anomalies: *Connector Envy*, *Scattered Parasitic Functionality* and *Component Concern Overload*.

Interestingly, the results show that architecture problems also occurred in the evolution history of systems or packages where architecture conformance was more strictly enforced in the code. The MIDAS project is the best example. Most architectural anomalies in MIDAS occurred due to interfaces are underlying the event-based middleware and misuse of connectors provided by the middleware. These anomalies were mostly cases of *Extraneous Connector* and *Connector Envy* occurrences. In addition, single components in MIDAS were realizing multiple scattered concerns, including service discovery, the fault tolerance policy, and dynamic adaptation. As a consequence, these components suffered from occurrences of *Component Concern Overload* and *Scattered Parasitic Functionality* anomalies.

As we can observe from the discussion above, code anomalies tend to manifest in different ways according to the system's characteristics. The extent of their contribution to either architectural erosion or architectural drift was also diverse. Regardless of these variations, the results revealed that a considerable amount of architectural modularity problems were introduced in the first system versions of all the 5 systems. This was observed even in MobileMedia, in which most of the architecture problems were introduced along the system evolution as discussed above. We further elaborate the implications of this finding in Section 6.

### 5.2 Accuracy of Investigated Detection Strategies

The accuracy of automated strategies for detecting architecturally-relevant code anomalies is summarized in Table 6. The token '-' is used in this table to represent the cases where modularity problems did not occur or they were not related to architectural problems. The average of the strategies' accuracy rates is also presented for anomalies in both object-oriented and aspect-oriented code. For aspect-

oriented systems, we concentrate on presenting the details related to the code anomaly occurrences. A detailed list of all code anomalies, false positives and false negatives in each one of the investigated systems can be found at [5].

In general, our analysis reveals that detection strategies are inaccurate in identifying architecturally-relevant code anomalies. Specifically, most of the automatically-detected code anomalies were not associated with architectural modularity problems, leading to many false positives. In general, the average of the automatically-detected code anomalies represented about 45% (or less) of the total number of code anomalies related to architectural modularity problems. MIDAS was the only exception, which will be discussed later. Consequently, these results might imply a problem to engineers who are interested in performing clean-up code revisions to avoid architecture degeneration. In these cases, developers are likely to devote most of their time analyzing code anomalies that do not represent a threat to the architecture modularity.

Even worse, many of the code anomalies harmful to architectural modularity problems were not automatically detected by strategies, leading to a high rate of false negatives. Developers will miss a wide range of architecture erosion and drift symptoms. In particular, many of the strategies exhibited recall rates close or much lower than 45%. That is, about 55% or more of the non automatically-detected code anomalies were related to architectural modularity problems. These results indicate that detection strategies seem to have a tendency to send developers in wrong directions when addressing code anomalies related to architectural modularity problem.

The next subsections discuss how accurate the strategies were when localizing code anomalies related to both violations (Section 5.2.1) and anomalies (Section 5.2.2).

### 5.2.1 Revealing Symptoms of Architecture Erosion

On average about 41% of the code anomalies related to violations were automatically-detected by strategies in the target systems. The results also show that code anomalies related to violations emerged in systems developed with both OO and AO modularity techniques. In OO systems, these violations were related to undesirable interdependencies between classes responsible for implementing different architectural elements. For instance, 69% of the violations in HealthWatcher were related to exception events propagated from the *DataManagement* layer to the *GUIElements* layer. Consequently, all interfaces between *DataManagement* and *GUIElements* layers propagated these exceptional events, even though the majority of these exceptions should be treated internally by classes defined in the *DataManagement* layer according to the designers. intent. The propagation of exception events introduced several occurrences of *Long Method*, *Misplaced Class*, *Divergent Change*, and *Shotgun Surgery*. However, just about 33% of these architecturally-relevant anomalies were automatically-detected by strategies.

Other kinds of violations emerged in AO systems as they follow a different architecture design. For instance 26% of the total number of architecturally-relevant anomalies was related to undesirable tight coupling between aspects and the base code. These relations were motivated by the fact that classes were exposing internal information just to be used by aspects. For instance, artificial methods had to be created in later system versions, aiming at allowing the expected composition between aspects. This situation leads to interface bloat occurrences and to the introduction of relevant *Long Parameter Lists* and *Forced Join Points*. However, detection strategies were able only to identify about 40% of these relevant occurrences.

### 5.2.2 Revealing Symptoms of Architecture Drift

Architectural anomalies were mostly related to the inappropriate modularization of architectural concerns in the target systems. *Exception Handling* for AspectualWatcher and *Connection* for AspectualMedia presented the strongest relationship with architectural modularity problems as they

Table 6. Results for the analyzed detection strategies

| Code Smells | True Positives | | | False Positives | | | False Negatives | | | Precision | | | Recall | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | HW | MM | MIDAS | HW | MM | MIDAS | HW | MM | MIDAS | HW | MM | MIDAS | HW | MM | MIDAS |
| Divergent Change | 7 | 1 | 4 | 14 | 2 | 43 | 19 | 2 | 2 | 0.33 | 0.33 | 0.09 | 0.27 | 0.33 | 0.67 |
| Feature Envy | 5 | 2 | - | 27 | 6 | - | 9 | 3 | - | 0.16 | 0.25 | - | 0.36 | 0.40 | - |
| God Class | 1 | 3 | 2 | 2 | 4 | 0 | 4 | 5 | 1 | 0.67 | 0.43 | 1.00 | 0.33 | 0.38 | 0.67 |
| Large Class | 1 | 1 | 2 | 2 | 0 | 4 | 4 | 1 | 0 | 0.43 | 1.00 | 0.30 | 0.38 | 0.50 | 1.00 |
| Long | 23 | 7 | 6 | 33 | 24 | 37 | 18 | 10 | 4 | 0.41 | 0.23 | 0.34 | 0.56 | 0.41 | 0.50 |
| Long Parameter List | 4 | - | - | 12 | - | - | 5 | - | - | 0.25 | - | - | 0.44 | - | - |
| Misplaced Class | 2 | 1 | - | 5 | 2 | - | 1 | 2 | - | 0.33 | 0.33 | - | 0.50 | 0.33 | - |
| Shotgun Surgery | 6 | 2 | 3 | 19 | 6 | 23 | 9 | 7 | 6 | 0.24 | 0.25 | 0.22 | 0.40 | 0.22 | 0.32 |
| **OO Avg. Rates** | | | | | | | | | | **0.35** | **0.40** | **0.33** | **0.41** | **0.38** | **0.63** |
| | **AW** | **AM** | **MIDAS** | **AW** | **AM** | **MIDAS** | **AW** | **AM** | **MIDAS** | **AW** | **AM** | **MIDAS** | **AW** | **AM** | **MIDAS** |
| **OO Avg. Rates** | | | | | | | | | | **0.47** | **0.32** | **-** | **0.38** | **0.44** | **-** |
| Composition Bloat | 2 | 3 | - | 4 | 1 | - | 3 | 4 | - | 0.33 | 0.50 | - | 0.40 | 0.43 | - |
| Duplicate Pointcut | 8 | 65 | - | 11 | 47 | - | 3 | 31 | - | 0.42 | 0.58 | - | 0.72 | 0.68 | - |
| Forced Join Point | 6 | 1 | - | 6 | 2 | - | 9 | 6 | - | 0.50 | 0.33 | - | 0.40 | 0.14 | - |
| God Aspect | 11 | 6 | - | 11 | 4 | - | 17 | 9 | - | 0.50 | 0.60 | - | 0.39 | 0.40 | - |
| God Pointcut | 10 | 8 | - | 20 | 7 | - | 14 | 11 | - | 0.33 | 0.53 | - | 0.42 | 0.42 | - |
| Redundant Pointcut | 52 | 3 | - | 17 | 3 | - | 32 | 2 | - | 0.75 | 0.50 | - | 0.62 | 0.60 | - |
| **AO Avg. Rates** | | | | | | | | | | **0.47** | **0.50** | **-** | **0.49** | **0.44** | **-** |

are very context-specific with code. *Exception Handling*, for instance, was scattered among different architectural components and, therefore, it was related to *Scattered Parasitic Functionality* occurrences. On the other hand, the high tangling of *Connection* with *Persistence* and *Logging* led to the architectural components responsible for its modularization were classified as *Component Concern Overload*. The inappropriate modularization of these concerns was associated with several occurrences of *Long Method*, *God Aspect*, *God Class*, *Divergent Change*, *Shotgun Surgery* in the target systems. *Exception Handling* and *Connection* were responsible, respectively, for 53 % and 41% of the total of architecturally-relevant code anomalies in AspectualWatcher and AspectualMedia. However, just about 47% of these relevant anomalies was automatically detected by strategies.

#### 5.2.3 Hypotheses and Overall Accuracy Results

Based on the aforementioned results, we can conclude that metrics-based strategies were not accurate in detecting architecturally-relevant code anomalies (Section 3.2). Therefore, we reject both null hypotheses $H1_0$ and $H2_0$ (Section 4.1) for all the systems, except MIDAS (Table 6). Several detection strategies presented recall rates greater than 60% in MIDAS. That is, more than a half of code anomalies related to architectural degradation symptoms were automatically identified by detection strategies in MIDAS. We also observed that the number architectural anomalies not related to code anomalies tend to increase compared with the other systems.

The MIDAS case confirmed our intuition that detection strategies are more effective in systems where architecture conformance is more strictly enforced in the code. The better the code modularity reflects the architecture decomposition, the fewer the number of code anomalies. This finding was not actually exclusive to MIDAS. Similar results were observed in packages of MobileMedia and HealthWatcher with highest adherence to the architectural rules. In these packages (e.g., Model for MobileMedia and Business for HealthWatcher) the detection strategies presented precision and recall rates higher than 60%. These packages also presented the lowest number of architecturally-relevant code anomalies.

Another relevant characteristic that is likely to favor the success of detection strategies (i.e., accuracy rates higher than 60%) is when the projection of architectural elements occurs in a few code units. In these cases, single code anomalies will exert a more direct impact on the architectural element that they are implementing. This phenomenon was observed in all target systems.

## 6. Analyzing Overlooked Code Anomalies

Once we have discussed the strategies' accuracy, we reflect upon the *key factors* that contributed to their failure in localizing architecturally-relevant code anomalies (Sections 6.1 and 6.2). This discussion can provide insights on how to improve the techniques to detect architecture degradation based on source code analysis.

### 6.1 Inability to Analyze Architectural Concerns' Properties in the Source Code

Code anomalies were often the source of architectural modularity problems when they were located in modules realizing various architectural concerns. We noticed that 62% of the total number of architecturally-relevant code anomalies exhibited this characteristic. This frequency reinforces that detection strategies should be more sensitive to the degree of concern scattering and tangling in the code. In fact, the employed strategies were not accurate when detecting anomalies associated with the inappropriate modularization of architectural concerns; they presented precision and recall rates around 43% and 48% respectively.

For instance, the class *BaseController* in MobileMedia was classified by developers as an architecturally-relevant occurrence of *God Class* since it is realizing different architectural concerns (e.g. *Photo*, *Music*, and *Persistence*). However, differently from *MediaController* (Figure 1), it was not automatically detected by the strategies. Even though this class was the source of highly tangled and scattered concerns, its methods present neither low cohesion nor high complexity (Section 3.1). However, changes associated with each of the architectural concerns were performed in this class, confirming its anomalous nature. This class was particularly related to two architectural anomalies, namely *Component Concern Overload* and *Scattered Parasitic Functionality*.

As a conclusion, the results reveal that conventional detection strategies are not accurate largely due to their lack of sensitivity to properties of architectural concerns in the code. Detection strategies are limited to metrics of structural properties (detected by static analysis tools) of modules in the code. Existing concern metrics [42] and concern tracing tools [10] should be leveraged to improve the accuracy of detection strategies used to assess architecture degradation.

### 6.2 Inability to Identify Architectural Information in the Source Code

Architecturally-relevant code anomalies often occurred in code elements responsible for implementing different architectural elements. Specifically, 49% of the architecturally-relevant code anomalies fell in this category. However, precision and recall rates of the strategies were 36% and 44%, respectively, when identifying these code anomalies.

For instance, the method *InsertEmployee.execute* in HealthWatcher represents an example of an architecturally-relevant code anomaly that was not automatically detected by our employed strategies. In particular, this method was classified as *Divergent Change* by developers since it accesses information and call methods of classes responsible for implementing different architectural elements. This method also introduces undesirable dependencies between

non-adjacent layers, condition to be classified as an architecturally-relevant occurrence. However, such *execute* method was not automatically detected by strategies because they focus on measuring method's strong coupling degree based on syntactic dependencies.

However, this method had instead a semantic dependency with other methods: the former changed together with other methods realizing different architectural components, which were not syntactically coupled to the former. Hence, we observed that strategies were not effective in detecting this kind of anomaly as they are not sensitive to which architectural elements a code anomaly is responsible for implementing. The key issue is that detection strategies cannot rely on information about how the code elements are associated with architectural modules and their inter-dependencies; this information cannot be extracted using code metrics. This might indicate the need for further investigating how detection strategies could exploit traces of architectural information in the code.

### 6.3 Patterns of Code Anomalies

It was observed that certain patterns of code anomalies tend to be better indicators of architectural degradation symptoms than single code anomalies. However, these patterns cannot be directly detected by strategies, which focus on identifying individual code anomalies. They do not capture, for instance, a chain of inter-related anomalies.

**Co-occurrences of Code Anomalies**. Certain recurring patterns of co-occurring code anomalies tend to be stronger indicators of architectural degradation symptoms. For instance, co-occurrences of *Long Method* and *Divergent Change* were associated with architectural problems in all the systems. That is, methods with either high cyclomatic complexity or many lines of code and, high coupling degree with different architectural elements were better indicators than single *Long Method* occurrences. More than 75% of these combined occurrences were associated with architectural problems while just about 43% of single *Long Method* occurrences were related to architectural problems.

It is important to point out that many of these relevant co-occurrences cannot be detected by simply combining multiple strategies using logical operators (Section 3.1). Aiming at identifying these co-occurrences, detection strategies must rely on some kind of architectural information (Section 6.2). For instance, it would be also useful to consider how many different architectural elements a method is accessing. Otherwise, strategies will just detect such relevant co-occurrences that present similar characteristics of non-relevant co-occurrences. That is, those co-occurrences that present tight coupling degree with several elements, disregarding their distribution on architectural decompositions.

**Code Elements suffering from the Same Anomaly**. Interesting findings emerged from analyzing *groups of code elements that suffer from the same code anomaly*. For instance, when a group of classes that suffer from *God Class* or *Large Class* are implementing the same architectural component *A* and realizing different concerns it may indicate that *A* suffers from *Component Concern Overload*. This assumption departs from the fact that *God Classes* and *Large Classes* are likely to be related to the inappropriate modularization of architectural concerns. Furthermore, when other architectural components and *God Classes* of *A* are sharing the same architectural concern, it may suggest that *A* is affected by *Scattered Parasitic Functionality*. This situation was observed in all the systems.

**Propagation of Architectural Problems**. It was also often observed the propagation of architectural problems from parents to children in the inheritance trees of all the systems. There are two main categories related to such *propagation of architectural problems*. The first is related to architectural problems that are propagated to all the children in the inheritance tree whereas in the second category the architectural problem is not propagated to all the children, i.e. some children are free of architectural problems. Examples of both categories were found in all systems. For instance, in HealthWatcher it was observed that several interfaces were introducing undesirable relationships via their parameter types. These interfaces were not identified by detection strategies because they had a well-defined interface (e.g. several members, without a high coupling degree). However, they had a considerable negative effect as these violations were propagated down through the class hierarchies. Usually these undesirable references are left in a system over a long period due to the ripple effects when refactorings are applied to remove them.

The limitations of detection strategies for localizing propagated relevant occurrences of code anomalies are the same for localizing single relevant occurrences. This is due to the propagation of code anomalies in the inheritance trees itself could be detected using static code analysis.

### 6.4 Architectural Design and Strategy Accuracy

There was a direct influence of the lack of modularity of certain concerns on the architecturally-relevant anomalies when analyzing different architectural decompositions. We observed that when the modularization of architectural concerns is more explicit in the source code the number of architecturally-relevant anomalies tend to decrease. For instance, OO systems presented a higher number of conventional code anomalies [13] than AO systems. We suspect this occurred due to most of the code anomalies were related to the inappropriate modularization of architectural concerns, which are more scattered in OO systems. As AOP mechanisms tend to improve the modularization of concerns in single aspects, they may remove relevant anomalies related to this factor. It is not our intention to compare the results in both decompositions, as we discussed in previous sections the inadequate use of AO mechanisms may introduce other kinds of architecturally-relevant code anomalies.

Even more interesting is the fact that we have observed how the strategies' accuracy for identifying architecturally-relevant anomalies seem to be similar in both kinds of architectural decompositions. This assumption is derived from results regarding to the "*average rows*" in Table 6. The strategies' accuracy rates are about 40% for detecting architecturally-relevant code anomalies in all AO and non-AO systems, except in MIDAS.

## 7. Threats to Validity

This section summarizes the main threats to validity and the mitigations considered; a detailed analysis of all the possible imperfections and mitigations for our study can be found at the supplementary website [5].

**Construct Validity**. Threats to construct validity are mainly related to possible errors introduced in the identification of code anomalies and architectural problems. There are different kinds of detection strategies documented in the literature. In particular, we opted for not selecting history-sensitive detection strategies as they tend to be less predictive and require multiple versions of the system [26, 40]. Consequently, they accurately reveal code anomalies just in later releases, when the system may have already achieved critical degradation stages.

We are aware that detection strategies, manual inspection and other mechanisms to identify code anomalies and architectural problems can introduce imprecision. However, we mitigated this threat by: (i) involving original developers and architects in this process, and (ii) using architectural models where architectural elements were mapped to different levels of granularity. That is, the relationships between code elements and architecture elements were often not 1-to-1. Furthermore, the architectural problems were identified by architects, who had previous experience on the detection of architectural violations and anomalies in other systems. The correlation analysis between code anomalies and architectural problems was also validated with the architects and developers.

**Conclusion Validity**. We have two issues that threaten the conclusion validity of our study: the number of evaluated systems and assessed anomalies. Two versions of MIDAS, eight versions of MobileMedia, eight versions of Aspectual-Media, ten versions of HealthWatcher and, ten versions of AspectualWatcher were used for the purposes of this study, totaling 38 versions. Of course, a higher number of systems is always desired. However, the analysis of a bigger sample in this study would be impracticable for different reasons.

First, the relationship between code anomalies and architectural problems needed to be confirmed by architects. Second, the number of systems with all the required information and stakeholders available to perform this study is rather scarce. Then, our sample can be seen as appropriate for a first exploratory investigation [20]. All the findings (for example, those discussed in Section 6) contribute with more specific hypotheses that should be further tested in repetitions or more controlled replications of our study.

Related to the second issue (completeness of code anomalies and architectural problems), our analysis was concerned with a wide variety of code anomalies and problems that occur in system's architecture. We analyzed the accuracy of detection strategies for identifying all architecturally-relevant code anomalies that occurred in the target systems. In addition, certain code anomalies were not discussed (e.g. *Data Class*) since their occurrences did not influence on studied system architectures.

**Internal and External Validity**. The main threats to internal and external validity are the following. First, the level of experience of systems' programmers could be an issue. In order to mitigate this, we used systems that were developed by more than 20 programmers with different levels of software development skills. The main threat to external validity is related to the nature of the evaluated systems. In order to minimize this threat we have tried to use applications with different sizes, that suffer from a different set of code anomalies and that were implemented using different architectural styles and environments. However, we are aware that more studies involving a higher number of systems should be performed in the future.

## 8. Concluding Remarks

Our results suggest that state-of-the-art detection strategies were not able to identify and locate architecturally relevant code anomalies. Specifically, more than 60% of the automatically-detected code anomalies were not correlated with architectural problems (neither with other threats, such as faults in the code). This means that developers might be spending a lot of time reviewing code anomalies (and refactoring code) that do not represent architectural (or other) threats to the system. Even worse, many of the false negatives (i.e. about 50%) generated by automated anomaly detection are often correlated with architectural problems. This means that developers would not be informed by detection strategies of code anomalies that are critical to architecture sustainability. These findings are interesting because they question the effectiveness of existing strategies and tools in supporting "architecture revision" strictly based on the source code (which is commonly the case). Also, it is in such case where the current mechanisms for "architecture revision" [1][8] cannot be used since they rely on the existence of the intended architectural design.

We found that the imperfection of the detection strategies is not simply related to specific thresholds or combinations of particular measures. On the contrary, the false positives and false negatives often cannot be resolved if design decisions are not traced and mapped to the source code, and exploited by detection strategies (Section 6). For instance, detection strategies cannot decide whether (or not) relationships between two classes are introducing violations. They cannot decide either whether a class is accessing information from classes defined in different architectural

elements. It was also found that certain recurring patterns of anomaly combinations or anomaly propagations are better indicators of architectural problems than individual anomaly occurrences. Therefore, developers should be warned about the harmful impact of these patterns and their existence in the source code in order to perform their early removal. However, these patterns usually cannot be specified or detected by existing techniques [21, 32], as they are intended to pick out individual anomaly occurrences.

## 9. Acknowledgements

This was sponsored by: I.Macia CNPq grant 579604/2008-0; A.Garcia FAPERJ grant E-26/102.211/2009, 111.152/2011 and CNPq grant 305526/2009-0; A.v.Staa CNPq grant 306802/2008-2; Projects: CNPq grants 483882/2009-7, 479344/2010-8 and 485348/2011-0. It was also sponsored by the US National Science Foundation under Grant number 1117593. Any opinions, findings, and conclusions expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF.

## References

[1] Aldrich, J. ArchJava: Connecting Software Architecture to Implementation. In Proc of the 24th ICSE, pp. 187-197, 2002.

[2] Alikacem, E.H and Sahraoui, H. Generic metric extraction framework. In Proc. of the 16th IWSM/MetriKon, 2006, pp. 383–390.

[3] Bieman, J.M. and Kang, B.K. Cohesion and Reuse in an Object Oriented System. In Proc of the ISSR, pp 259-262, 1995.

[4] Clements, P et al. Documenting Software Architectures: Views and Beyond. Addison-Wesley, 2nd Edition, 2010

[5] Code smells study: http://www.inf.puc-rio.br/~ibertran/aosd12.

[6] D'Ambros, M. et al. the Impact of Design Flaws on Software Defects. In Proc. of the 10th QSIC, pp. 23 - 31, 2010.

[7] Dhambri et al. Visual Detection of Design Anomalies. In Proc. of the 12th CSMR, pp. 279-283, 2008.

[8] Eichberg, M. et al. Defining and Continuous Checking of Structural Program Dependencies. In Proc. of the 30th ICSE, 2008.

[9] Emden, E. and Moonen, L. Java quality assurance by detecting code smells. In Proceedings of the 9th ICRE, 2002.

[10] FEAT tool, http://www.cs.mcgill.ca/~swevo/feat/

[11] Ferrari, F. et al. An exploratory study of error-proneness in evolving Aspect-Oriented Programs. In: Proc. of the 25th OOPSLA, USA, 2009.

[12] Figueiredo, E. et al. Evolving software product lines with aspects: An empirical study on design stability. In Proc of the 30th ICSE, 2008.

[13] Fowler, M. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[14] Garcia, J. et al. Identifying architectural bad smells. In Proc of the. 13th CSMR, pp 255–258, 2009.

[15] Greenwood, P. et al. On the impact of aspectual decompositions on design stability: An empirical study. In Proc. of the 21st ECOOP, 2007.

[16] Hochstein, L. and Lindvall, M. Combating architectural degeneration: A survey. Info. & Soft. Technology July, 2005.

[17] Hosmer, D. and Lemeshow, S. Applied Logistic Regression (2nd Edition). Wiley, 2000.

[18] Khomh, K. et al. An exploratory study of the impact of code smells on software change-proneness. In Proc of the 16th WCRE, 2009.

[19] Kiczales, G.,et al. Aspect-oriented programming. In Proc. of the 11th ECOOP. LNCS, vol. 1241. Springer, Heidelberg. pp. 220-242, 1997.

[20] Kitchenham, B. et al. Evaluating guidelines for empirical software engineering studies. ISESE pp 38-47, 2006

[21] Lanza, M. and Marinescu, R. Object-Oriented Metrics in Practice. Springer, 2006.

[22] Lippert, M. and Roock, S. Refactoring in Large Software Projects: Performing Complex Restructurings Successfully. Wiley. 2006.

[23] Macia, I. et al. A. An Exploratory Study of Code Smells in Evolving Aspect-Oriented Systems. In Proc of the 10th AOSD, 2011.

[24] Malek, S. et al. Reconceptualizing a family of heterogeneous embedded systems via explicit architectural support. In Proc. of the 29th ICSE. 2007.

[25] Mantyla, M.V. and Lassenius, C. Subjective evaluation of software evolvability using code smells: An empirical study. Empirical Software Enggineering, vol. 11, no. 3, pp. 395–431, 2006.

[26] Mara, L. et al. Hist-Inspect: A Tool for History-Sensitive Detection of Code Smells. In Proc. of the 10th AOSD, 2011

[27] Marinescu, R. Detection strategies: Metrics-based rules for detecting design flaws. In Proc. of the 20th ICSM, pp 350-359, 2004.

[28] Marinescu,R.; Ganea, G. and Veredi, I. inCode: Continuous Quality Assessment and Improvement. In Proc of the 14th CSMR, 2010.

[29] Martin, R. Agile Principles, Patterns, and Practices. Prentice Hall, 2002.

[30] McCabe, T.J. A Software Complexity Measure. IEEE Transactions on Software Engineering, 2 (4), pp 308-320, 1976.

[31] Meyer, B. Object-Oriented Software Construction. Prentice Hall Professional Technical 2nd edition, 2000.

[32] Moha, N. et al. DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE TSE, 2010.

[33] Munro, MJ. Product metrics for automatic identification of bad smell design problems in java source-code. In Proc of 11th METRICS, 2005

[34] MuLATo tool, http://sourceforge.net/projects/mulato/ (3/08/2009)

[35] Murphy, G.C., et al.. Software Reflexion Models: Bridging the Gap between Design and Implementation. IEEE TSE, pp 364–380, 2001.

[36] Murphy-Hill, E. Scalable, expressive, and context-sensitive code smell display. In Proc of the 23rd OPSLA, 2008.

[37] Olbrich, S.M. et al. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In Proc of the 26th ICSM pp 1-10, 2010.

[38] Olbrich, S.M. et al. The evolution and impact of code smells: A case study of two open source systems. In Proc of the 3rd ESEM, 2009.

[39] Perry, D.E. and Wolf, A.L. Foundations for the study of software architecture, ACM Software. Eng. Notes 17 (4) pp 40–52, 1992.

[40] Ratiu, D. et al. Using History Information to Improve Design Flaws Detection. In Proc of the 8th CSMR, 2004.

[41] Ratzinger, J. et al. Improving evolvability through refactoring. In Proc of the 5th IEEE MSR, 2005.

[42] Sant'anna, C. et al. On the modularity of software architectures: A Concern-Driven measurement framework. In Proc. of ECSA, 2007.

[43] Sonar: http://docs.codehaus.org/display/SONAR/

[44] Srivisut, K. and Muenchaisri, P. Bad-smell Metrics for Aspect-Oriented Software. In Proc of the 6th ICIS, 2007.

[45] Together: http://www.borland.com/us/products/together/

[46] Tsantalis, N. and Chatzigeorgiou, A. Identification of move method refactoring opportunities. IEEE TSE, 35(3), pp 347–367, 2009.

[47] Understand: http://www.scitools.com/

[48] Wake, W.C. Refactoring Workbook. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[49] Wettel, R. and Lanza, M. Visually localizing design problems with disharmony maps. In Proc. of the 4th Softvis pp. 155–164, 2008.